

The MDS Queue

Nihar B. Shah, Kangwook Lee, Kannan Ramchandran
 Dept. of Electrical Engineering and Computer Sciences
 University of California, Berkeley
 {nihar, kw1jjang, kannanr}@eecs.berkeley.edu

Abstract

In order to scale economically, data centers are increasingly evolving their data storage methods from the use of simple data replication to the use of more powerful erasure codes, which provide the same level of reliability as replication-based methods at a significantly lower storage cost. In particular, it is well known that Maximum-Distance-Separable (MDS) codes, such as Reed-Solomon codes, provide the maximum storage efficiency. While the use of codes for providing improved reliability in archival storage systems, where the data is less frequently accessed (or so-called “cold data”), is well understood, the role of codes in the storage of more frequently accessed and active “hot data” is less clear. In fact, a key question is: when the performance metric is no longer data reliability but rather *latency*, do codes even help?

In this paper, we answer this question in the affirmative by studying coded data storage systems based on MDS codes through the lens of queueing theory, and term this the “MDS queue.” We analytically characterize the latency performance of MDS queues, and reveal its superior performance (up to 70%) compared to that of currently used replication-based schemes. In our analysis of MDS queues, we present insightful scheduling policies that form upper and lower bounds to performance, and show that they are quite tight. Extensive simulations of the MDS queue using Markov-Chain-Monte-Carlo (MCMC) methods are also provided and used to validate our theoretical analysis. As a side note, our lower-bound analytical method based on the so-called MDS-Reservation(t) queue, represents an elegant practical scheme that requires the maintenance of considerably smaller state, depending on the parameter t , than that of the full-fledged MDS queue (which corresponds to $t = \infty$), and may be of independent interest in practical systems.

I. INTRODUCTION

Two of the primary objectives of a storage system are to provide reliability and availability of the stored data: the system must ensure that data is not lost even in the presence of individual component failures, and must be easily accessible to the user whenever required. The classical means of providing reliability is to employ the strategy of *replication*, wherein identical copies of the (entire) data are stored on multiple servers. However, this scheme is not very efficient in terms of the storage space utilization. The exponential growth in the amount of data being stored today makes storage an increasingly valuable resource, and has motivated data-centers today to increasingly turn to the use of more efficient storage codes [1]–[4].

Fig. 1 provides an illustrative example comparing the replication and coded schemes. Here, four files F_A , F_B , F_C and F_D , are to be stored in a reliable manner across four servers. The replication strategy, as depicted in Fig. 1a, stores $\{F_A, F_B\}$ in the first two servers and $\{F_C, F_D\}$ in the two remaining servers. On the other hand, the strategy of coding as shown in Fig. 1b, partitions each file into two halves as $F_x = [f_{x1} \ f_{x2}]$ for $x \in \{A, B, C, D\}$, and stores the four sets $\{f_{x1}\}$, $\{f_{x2}\}$, $\{f_{x1} + f_{x2}\}$, and $\{f_{x1} + 2f_{x2}\}$ in four different servers. Observe that the total storage space required under this coding scheme is identical to that required under the replication scheme. However, while the replication scheme loses files F_A and F_B upon failure of the first two servers, and loses files F_C and F_D upon failure of the two other servers, the coding scheme loses no data even upon failure of

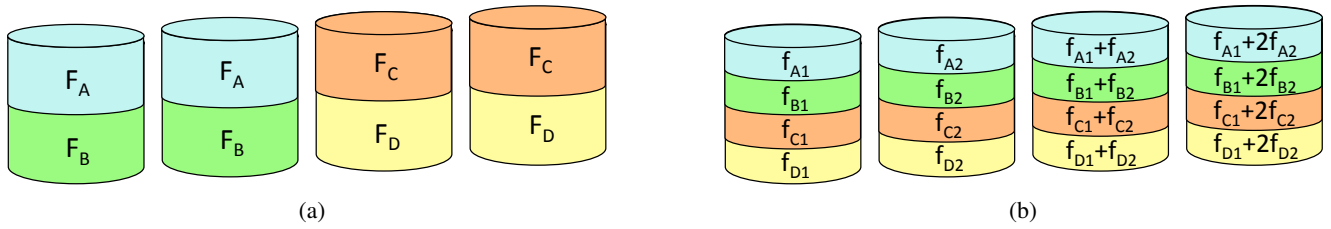


Fig. 1: An example comparing storage schemes based on (a) replication and (b) (MDS) codes. The data to be stored consists of four files F_A , F_B , F_C and F_D . Each of these files are split into two halves in the coded system as $F_x = \{f_{x1}, f_{x2}\}$, for $x \in \{A, B, C, D\}$. Observe that in the coded system, any file can be recovered by reading data from *any* two of the four servers.

any two of the four servers. Thus, the coding scheme in this example provides a higher reliability as compared to replication, for the same level of storage efficiency. While this was only a toy example, the amount of gains offered by codes can be significantly higher when the file is split into a larger number of chunks. Furthermore, any redundancy scheme based on replication must store at least one additional copy of the entire data, which necessitates a storage overhead of at least 100%. This may itself be prohibitive in many cases. On the other hand, codes do not face such a barrier and can support smaller overheads.

The most popular, and also most efficient storage codes are a class of codes known as Maximum-Distance-Separable (MDS) codes [5]. The Reed-Solomon code [6] is an example of an MDS code, and so is the code depicted in Fig. 1b. An MDS code is typically associated to two parameters n and k . Under an (n, k) MDS code, a file is encoded and stored in n servers such that (a) the data stored in *any* k of these n servers suffice to recover the entire file, and (b) the storage space required at each server is $\frac{1}{k}$ of the size of the original file.¹

As a result of the storage savings they offer, codes have been employed in data centers to store ‘cold’ data, i.e., data that is less frequently accessed, where reliability and storage efficiency are the primary metrics. While this application of codes is fairly well studied, the performance of codes for the storage of more frequently accessed “hot data” is less clear. In this context, a key question is: when the performance metric is no longer data reliability but rather *latency*, do codes even help?

In this paper, we answer this question in the affirmative by studying coded data storage systems based on MDS codes through the lens of queueing theory, and term the queue resulting out of the use of MDS codes as the *MDS queue*. We analytically characterize the latency performance of MDS queues, and reveal its superior performance (up to 70%) as compared to that of currently used replication-based schemes, under the models considered in this paper. In our analysis of MDS queues, we also present insightful scheduling policies that form upper and lower bounds to performance, and show that they are quite tight. Extensive Markov-Chain-Monte-Carlo (MCMC) simulations of the MDS queue are also provided and used to validate our theoretical analysis.

A. Definition of the MDS queue

We shall now describe the queueing theoretic model of a system employing an MDS code. To arrive at this model, we first make some simplifying assumptions in the interests of conceptual simplicity. We assume homogeneity among files, among requests, and among servers: the files are of identical size, incoming requests

¹A more rigorous definition of an MDS code is that it is a code that satisfies the ‘Singleton bound’ [5].

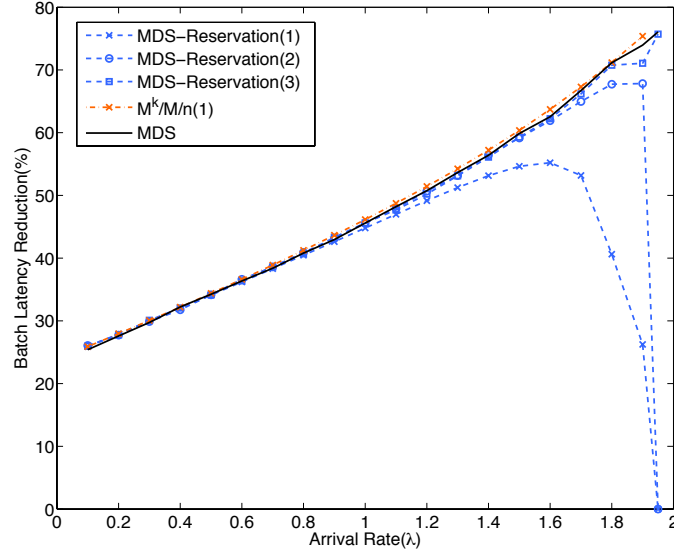


Fig. 2: The percentage reduction in average latency of a coded system over a replication-based system, for a storage system with parameters $n = 10$, $k = 5$ and service rate $\mu = 1$. The curve titled ‘MDS’ is the reduction achieved by the exact coded system. Also plotted are the gains achieved by the lower bounds (MDS-Reservation(t) queues) and upper bounds ($M^k/M/n(t)$ queues) presented in this paper. The graphs plotted are from MCMC simulations, which closely match the analytical evaluations of the lower and upper bounds. The fall in the gains achieved by some of the lower bounds at high arrival rates is because of their lower throughput, which makes the queues associated with the bounds unstable at arrival rates close to their maximum throughput.

are distributed as a Poisson process independent of the state of the system, and every incoming request asks for reading out one particular file with the reading process at every server following an independent and identical exponential distribution.²

As discussed previously, under an MDS code, a file can be retrieved by downloading data from any k of the servers. We model this by treating each request for reading a file as a *batch* of k jobs. The k jobs of a batch represent the reads of k parts of the file from k servers. A batch is considered as served when all k of its jobs have completed service. For instance, a request for reading file F_A in the system depicted in Fig. 1b is treated as a batch of two jobs. To service this request, the two jobs may be served by any two of the four servers; for example, if the two jobs are served by servers 2 and 3, then they correspond to reading f_{A2} and $(f_{A1} + f_{A2})$ respectively, which suffices to obtain the desired file F_A .

Definition 1 (MDS queue): An MDS queue is associated to four parameters (n, k) and $[\lambda, \mu]$.

- There are n identical servers
- Requests enter into a (common) buffer of infinite capacity
- Requests arrive in *batches* of k jobs each
- These batches arrive as a Poisson process with a rate of λ
- Each of the k jobs in a batch can be served by an **arbitrary** set of k **distinct** servers

²While the service times in practice are unlikely to be distributed exponentially, such an assumption is meant to serve as a starting point for more rigorous analyses.

- The service time for a job at any server is exponentially distributed with a rate of μ , and is independent of the arrival and service times of all other jobs
- The jobs are processed in order, i.e., among all the waiting jobs that an idle server is allowed to serve, it serves the one which had arrived the earliest.

The scheduling policy that governs this queue is formalized in Algorithm 1.

Algorithm 1 MDS scheduling policy

```

on arrival of batch
  assign as many jobs (of the new batch) as possible to idle servers
  append the remaining jobs (if any) as a new batch at the end of the buffer
on departure from a server (say, server  $s$ )
  if  $\exists$  at least one batch in the buffer such that no job of this batch has been served by  $s$  then
    among all such batches, find the batch that had arrived earliest
    assign a job from this batch to  $s$ 
  end if

```

The following example illustrates the functioning of the MDS scheduling policy and the resultant MDS queue.

Example 1: Consider the MDS($n=4, k=2$) queue, as depicted in Fig. 3. Here, each request comes as a batch of $k=2$ jobs, and hence we denote each batch (e.g., A, B, C , etc.) as a pair of jobs (e.g., $\{A_1, A_2\}$, $\{B_1, B_2\}$, $\{C_1, C_2\}$, etc.). The two jobs in a batch need to be served by (any) two distinct servers. Denote the four servers (from left to right) as servers 1, 2, 3 and 4. Suppose the system is in the state as shown in Fig. 3a, wherein the jobs A_2, A_1, B_1 and B_2 are being processed by the four servers, and there are three more batches waiting in the buffer. Suppose server 1 completes servicing job A_2 (Fig. 3b). Now, this server is free to serve any of the 6 jobs waiting in the buffer. However, since we allow jobs to be processed only in order, server 1 begins servicing job C_1 (assignment of C_2 instead would also have been valid). Next, suppose the first server completes service of C_1 before any other servers complete their current tasks (Fig. 3c). In this case, since server 1 has already served a job from the $\{C_1, C_2\}$ batch, it is not allowed to service C_2 (due to the restriction in the 5th bullet of the definition of the MDS queue). However, it can service any job from the next batch $\{D_1, D_2\}$, and one of these two jobs is (arbitrarily) assigned to it. Finally, when one of the other servers completes its service, then that server is assigned to service C_2 (Fig. 3d).

It turns out that a special case of the MDS queue, when $k=1$, is equivalent to an M/M/n queue. The MDS queue for a general value of k differs from the M/M/n queue in two ways. In the MDS(n, k) queue,

- jobs arrive in batches of k , and
- each job in a batch must be served by a different server.

An exact analysis of the MDS queue is hard: a Markov chain representation to keep track of only the configuration of the buffer requires a state space that is infinite in k dimensions. Thus, in this paper, we provide scheduling policies for the MDS queue that lower/upper bound its exact performance.

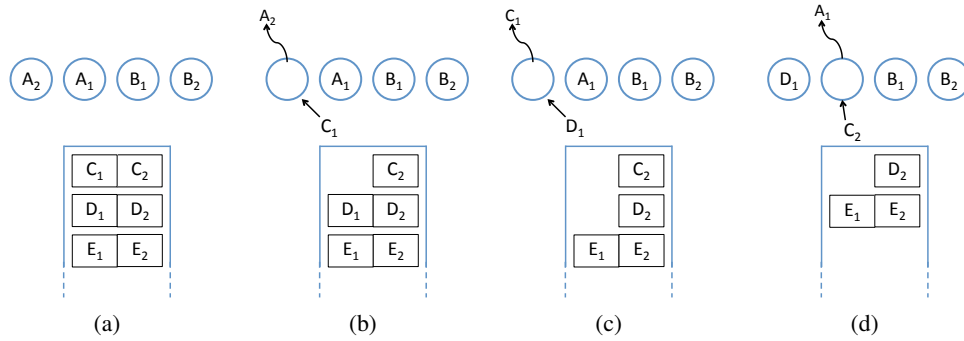


Fig. 3: Illustration of the functioning of the MDS queue. Requests take the form of batches, A , B , C etc., each comprising of $k=2$ jobs $\{A_1, A_2\}$, $\{B_1, B_2\}$, $\{C_1, C_2\}$, etc. The two jobs of any batch can be served by any two distinct servers.

B. Summary of results and contributions

In this paper, we undertake a queueing theoretic study of the performance of coded systems with respect to the key metric of latency. We model the system as a queue, which we term the “MDS queue.”

We provide scheduling policies to bound the performance of the MDS queue. These scheduling policies are chosen so as to capture the most significant features of the MDS queue, while allowing analytical tractability. Analysing these scheduling policies in terms of several performance metrics, we provide tight bounds on the performance of the MDS queue. In addition, we also provide Markov-Chain-Monte-Carlo (MCMC) simulations for the MDS queue, which are used to validate our analytical results, and also target other metrics such as the tails of the distribution of the latency that are hard to evaluate otherwise.

Our analysis and simulations reveal that codes can achieve significant gains over replication based systems, under the models considered in this paper. For example, the average latency incurred by a read-request in a coded system may enjoy upto a 70% reduction in latency on an average as compared to replication based schemes, as shown in Fig. 2; the 99th percentile tails of the latency may reduce by upto 50%, as shown subsequently in the paper. *The key insight is that the property of being able to recover the data from any k of the servers that lends MDS codes a high storage efficiency, also allows for greater parallel access, and is thus instrumental in providing it a low latency.* This paper quantifies these gains.

In addition to analysing the performance of coded systems with respect to several standard metrics, we also employ the framework presented in this paper to study additional topics of interest, such as comparing various methods of performing degraded reads, and understanding the effect of flooding requests.

The lower bounds (the $MDS\text{-}Reservation(t)$ scheduling policies) and the upper bounds (the $M^k/M/n(t)$ scheduling policies) are both indexed by a parameter ‘ t ’. An increase in the value of t results in tighter bounds, but also increases the complexity of analysis. Furthermore, both classes of scheduling policies converge to the MDS scheduling policy as $t \rightarrow \infty$. We note that the $MDS\text{-}Reservation(t)$ scheduling policies presented here are themselves a practical alternative to the MDS scheduling policy, since they require maintenance of a smaller state, while offering highly comparable performance. While the performance as well as the complexity of the $MDS\text{-}Reservation(t)$ queue increases with an increase in the value of the parameter t , our analysis shows that its performance is very close to that of the MDS queue for very small values of t (as small as $t=3$). Likewise, the performance of the upper bounds closely follow that of the MDS queue for values of t as small as $t=1$.

To allow the reader to evaluate performance of these queues under any parameters of choice, the source code of the MCMC simulations as well as those for computing steady-state distributions and performance metrics of the proposed queues have been made freely downloadable from the websites of the first two authors.

The remainder of the paper is organized as follows. Section II provides a background on this problem and presents related literature. Section III describes the general approach and the notation followed in the paper. Section IV presents the MDS-Reservation(t) queues, that lower bound the performance of the MDS queue. Section V presents the $M^k/M/n(t)$ queues, that are upper bounds to the performance of the MDS queue. Section VI presents an analysis and comparison of each of these queues, and queues resulting from replication-based storage schemes. The paper concludes with a discussion in Section VII. The appendix contains proofs of the theorems presented in the paper.

II. RELATED LITERATURE

A. Latency and storage efficiency in data-centers

1) *Latency analysis*: The study of latency comparisons between replication and coded systems was initiated by Huang et al. in [7], which we build upon in this paper. They presented a scheduling policy termed the ‘block-one-scheduling’ policy, that provides a lower bound on the performance of the MDS queue. They analysed this policy for $k=2$ and showed that this scheme improves the average latency faced by a job by upto 17% as compared to the replication scheme. The block-one-scheduling policy is a special case of the MDS-Reservation(t) queues presented in this paper, and corresponds to the case when $t=1$. While the block-one-scheduling policy was analysed for $k=2$ in [7], the analysis in this paper presents applies to any values of (n,k) , and recovers the corresponding results of [7] as a special case. In addition, the MDS-Reservation(t) queues of this paper, when $t \geq 2$, provide considerably tighter bounds to the performance of the MDS queue.

2) *Blocking probability*: The blocking probability of the MDS queue in the absence of a buffer was previously studied in [8], for arbitrary service time distributions. In this paper, as a special case of the MDS queue analysis, we study the blocking probabilities for the case when there is no buffer as well as for the case when there is a buffer with a finite capacity. However, in this paper, we restrict our attention to the case when the service times are exponentially distributed.

3) *Request-flooding*: One possible means of reducing latency is to send any request (redundantly) to more servers than necessary, and collecting the results from the first of the servers to respond. Such ‘request-flooding’ policies have been often suggested in the literature [9]–[11] for various applications. In particular, Joshi et al. [11] consider a setting very similar to the one considered in this paper. They propose a request-flooding policy wherein a batch would be sent to *all* n servers, and upon completion of any k of the n jobs, the remaining jobs would be cancelled. They provide bounds on the average latency faced by a batch in the steady-state. However, request-flooding to all n servers may be relatively resource expensive, and furthermore, the cancellation of jobs would typically mandate a certain cost (which is not accounted for in the models considered in [11]).

In this paper, we introduce an additional *flooding parameter*, ρ . In this new setting, a batch may be served by upto ρ ($\geq k$) arbitrary servers, and the batch is considered as served whenever any k of these servers complete service. In addition, we assume that the cancellation of a job at a server requires the server to remain idle for a certain time, accounting for the cost incurred upon the cancellation. In addition to the rigorous analysis for $\rho=k$ that forms a bulk of this paper, we perform MCMC simulations to evaluate and compare the system performance for various values of the flooding parameter ρ .

4) *Degraded Reads*: The coded system presented so far assumes that each incoming request desires one complete file F_x , for some $x \in \{A, B, C, \dots\}$. However, in certain applications, some incoming requests may require only a part of the file, say, $f_{x\ell}$ for some $\ell \in \{1, \dots, k\}$. Let us assume that the code employed is systematic: servers $1, 2, \dots, k$ respectively store the data $f_{x1}, f_{x2}, \dots, f_{xk}$. In this case, a request for $f_{x\ell}$ can be served by reading $f_{x\ell}$ directly from the ℓ^{th} server. In the event that server ℓ is busy or unavailable, $f_{x\ell}$ will have to be recovered from the data stored in the remaining $(n-1)$ servers. Such an operation is called a ‘degraded read’.

Under any MDS code, a degraded read may be performed by downloading the data stored in any k of the remaining $(n-1)$ servers to obtain F_x , and then extracting the desired chunk $f_{x\ell}$ from F_x . An alternative means of performing degraded reads has been proposed recently in [12]–[14]. In particular, practical MDS codes have been proposed in [14] (termed the ‘product-matrix codes’) that are associated to an additional parameter d , and can recover $f_{x\ell}$ of file F_x by reading and downloading small fractions of the data stored in *any* d of the remaining $(n-1)$ servers. In this paper, we employ our queueing theoretic framework analyse and compare the latency-performance of these two methods of performing degraded reads.

B. Diversity and error-correction

The MDS queue also arises in other applications that require diversity or error correction. For instance, consider a system with n processors, with jobs arriving as a Poisson process. It is often the case that the processors are not completely reliable [15], and may give incorrect outputs at random times. In order to guarantee a correct output, a job may be processed at k different servers, and the results aggregated (perhaps by a majority rule) to obtain the final answer. Such a system results precisely in an MDS(n, k) queue. In general, queues where jobs require diversity, for purposes such as security, error-protection etc., may be modelled as an MDS queue.³

C. Fork-join queues for parallel processing

A class of queues that are closely related to the MDS queue are the *fork-join queues* [16], [17]. A fork-join queue is used to model systems wherein jobs require multiple resources that can be provided in parallel. These queues are similar to the MDS queue in the sense that in both queues, jobs arrive in batches of k , and each job needs to be served by one distinct server out of the n servers. However, the distinction between these two classes of queues is that under a fork-join queue, each job must be served by a particular *pre-specified* server, while under an MDS queue, the k servers serving the jobs can be chosen by the scheduling policy. It thus follows that the performance of an MDS(n, k) queue is lower bounded by that of an (n, k) fork-join queue, with the gap quantifying the gains due to the flexibility in scheduling. Furthermore, as a special case, the MDS(n, n) queue is identical to the (n, n) fork-join queue.

III. OUR APPROACH AND NOTATION

For each of the scheduling policies presented in this paper (for lower/upper bounding the MDS queue), we represent the respective resulting queues as continuous time Markov chains. We show that these Markov chains belong to a class of processes known as *Quasi-Birth-Death (QBD)* processes (described below), and obtain their steady-state distribution by exploiting the properties of QBD processes.

³An analogy that the academic will relate to is that of reviewing papers. There are n reviewers in total, and each paper must be reviewed by k reviewers. This forms an MDS(n, k) queue. The values of λ and μ should be chosen such that $\frac{\lambda}{\mu}$ is close to the maximum throughput, modelling the fact that reviewers are generally busy.

Throughout the paper, we shall refer to the entire setup described in Section I-A as the ‘queue’ or the ‘system’. We shall say that a batch is waiting (in the buffer) if at-least one of its jobs is still waiting in the buffer (i.e., has not begun service). We shall use the term “ i^{th} waiting batch” to refer to the batch that was the i^{th} earliest to arrive, among all batches currently waiting in the buffer. For example, in the system in the state depicted in Fig. 3a, there are three waiting batches: $\{C_1, C_2\}$, $\{D_1, D_2\}$ and $\{E_1, E_2\}$ are the first, second and third waiting batches respectively.

Table I enumerates notation for various parameters that describe the system at any given time. To illustrate this notation, consider again the system depicted in Fig. 3a. Here, the parameters listed in Table I take values $m = 10$, $z = 0$, $b = 3$, $s_1 = 0$, $s_2 = 0$, $s_3 = 0$, $w_1 = 2$, $w_2 = 2$ and $w_3 = 2$. One can verify that keeping track of these parameters leads to a valid Markov chain (under each of the scheduling policies discussed in this paper). Note that we do not keep track of the jobs of a batch once all k jobs of that batch have begun to be served, nor do we track what servers are serving what jobs. This is to ensure a smaller complexity of representation and computation. Further note that in terms of the parameters listed in Table I, the number of servers that are busy at any given time is equal to $(n - z)$. For batch i in the buffer ($i \in \{1, \dots, b\}$), the number of jobs that have completed service is equal to $(k - s_i - w_i)$. For any integer i , $w_i = 0$ will mean that there is no i^{th} waiting batch in the buffer.

We shall frequently refer to an MDS queue as MDS(n, k) queue, and assume $[\lambda, \mu]$ to be some fixed (known) values. The system will always be assumed to begin in a state where there are no jobs in the system (i.e., with $m = b = 0$). Since the arrival and service time distributions have valid probability density functions, we shall assume that no two events occur at precisely the same time. We shall use the notation a^+ to denote $\max(a, 0)$.

Review of Quasi-Birth-Death (QBD) processes: Consider a continuous-time Markov process on the states $\{0, 1, 2, \dots\}$, with transition rate λ_0 from state 0 to 1, λ from state i to $(i+1)$ for all $i \geq 1$, μ_0 from state 1 to 0, and μ from state $(i+1)$ to i for all $i \geq 1$. This is a birth-death process. A QBD process is a generalization of such a birth-death process, wherein, each state i of the birth-death process is replaced by a set of states. The states in the first set (corresponding to $i=0$ in the birth-death process) is called the set of *boundary* states, whose behaviour is permitted to differ from that of the remaining states. The remaining sets of states are called the *levels*, and the levels are identical to each other (recall that all states $i \geq 1$ in the birth-death process are identical). The Markov chain may have transitions only within a level or the boundary, between adjacent levels, and between the boundary and the first level. The transition probability matrix of a QBD process is thus of the

Value	Meaning	Range
m	number of jobs in the entire system	0 to ∞
z	number of idle servers	0 to n
b	number of waiting batches	0 to ∞
$\{s_i\}_{i=1}^b$	number of jobs of i^{th} waiting batch, in the servers	0 to $k-1$
$\{w_i\}_{i=1}^b$	number of jobs of i^{th} waiting batch, in the buffer	0 to k

TABLE I: Notation used to describe state of the system.

form

$$Q = \begin{bmatrix} B_1 & B_2 & 0 & 0 & 0 & \cdots \\ B_0 & A_1 & A_2 & 0 & 0 & \cdots \\ 0 & A_0 & A_1 & A_2 & 0 & \cdots \\ 0 & 0 & A_0 & A_1 & A_2 & \cdots \\ 0 & 0 & 0 & A_0 & A_1 & \cdots \\ 0 & 0 & 0 & 0 & A_0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \quad (1)$$

Here, the matrices B_0 , B_1 , B_2 , A_0 , A_1 and A_2 represent transitions entering the boundary from the first level, within the boundary, exiting the boundary to the first level, entering a level from the next level, within a level, and exiting a level to the next level respectively. If the number of boundary states is q_b , and if the number of states in each level is q_ℓ , then the matrices B_0 , B_1 and B_2 have dimensions $(q_\ell \times q_b)$, $(q_b \times q_b)$ and $(q_b \times q_\ell)$ respectively, and each of A_0 , A_1 and A_2 have dimensions $(q_\ell \times q_\ell)$. The birth-death process described above is a special case with $q_b = q_\ell = 1$ and $B_0 = \mu_0$, $B_1 = 0$, $B_2 = \lambda_0$, $A_0 = \mu$, $A_1 = 0$, $A_2 = \lambda$. Figures 5, 7 and 10 in the sequel also present examples of QBD processes.

QBD processes are very well understood [18], and their stationary distribution is fairly easy to compute. In this paper, we employ the *SMCSolver* software package [19] for this purpose. In the next two sections, we present scheduling policies which lower and upper bound the performance of the MDS queue, and show that the resulting queues can be represented as QBD processes. This property makes them easy to analyse, and this is exploited subsequently in the analysis presented in Section VI.

IV. THE MDS-RESERVATION(T) QUEUES: LOWER BOUNDS

We now present a class of scheduling policies (and the resulting queues), which we call the MDS-Reservation(t) scheduling policies (and MDS-Reservation(t) queues), whose performance lower bounds the performance of the MDS queue. We shall see subsequently that the MDS-Reservation(t) scheduling policies are practical policies that require maintenance of a much smaller state as compared to the MDS queue, and hence may be of independent interest. This class of scheduling policies are indexed by a parameter ‘t’: a higher value of t leads to a better performance and a tighter lower bound to the MDS queue, but on the downside, requires maintenance of a larger state and is also more complex to analyse.

The MDS-Reservation(t) scheduling policy, in a nutshell, is as follows:

“apply the MDS scheduling policy, but with an additional restriction that for any $i \in \{t+1, t+2, \dots\}$, the i^{th} waiting batch is allowed to move forward in the buffer only when all k of its jobs can move forward together.”

We first describe in detail the special cases of $t=0$ and $t=1$, before moving on to the scheduling policy for a general t .

A. MDS-Reservation(0)

1) *Scheduling policy*: The MDS-Reservation(0) scheduling policy is rather simple: the batch at the head of the buffer may start service only when k or more servers are idle. The policy is described formally in Algorithm 2.

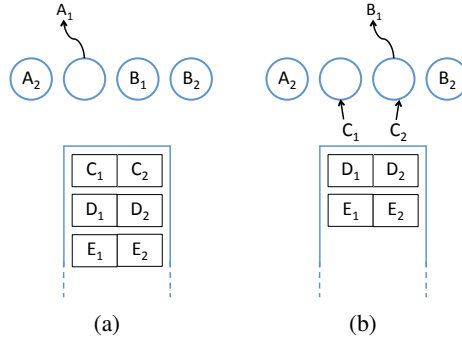


Fig. 4: An illustration of the MDS-Reservation(0) scheduling policy for a system with parameters $(n=4, k=2)$. This policy prohibits the servers to process jobs from a batch unless there are k idle servers that can process all k jobs of that batch. As shown in the figure, server 1 is barred from processing $\{C_1, C_2\}$ in (a), but is subsequently allowed to do so when another server also becomes idle in (b).

Algorithm 2 MDS-Reservation(0) Scheduling Policy

```

on arrival of a batch
  if number of idle servers  $< k$  then
    append new batch at end of buffer
  else
    assign  $k$  jobs of the batch to any  $k$  idle servers
  end if
on departure from server
  if (number of idle servers  $\geq k$ ) and (buffer is non-empty) then
    assign  $k$  jobs of the first waiting batch to any  $k$  idle servers
  end if

```

Example 2: Consider the MDS($n=4, k=2$) queue in the state depicted in Fig. 3a. Suppose the server 2 completes processing job A_1 (Fig. 4a). Upon this event, the MDS scheduling policy would have allowed server 2 to take up execution of either C_1 or C_2 . However, this is not permitted under MDS-Reservation(0), and this server remains idle until a total of at least $k=2$ servers become idle. Now suppose the third server completes execution of B_1 (Fig. 4b). At this point, there are sufficiently many idle servers to accommodate all $k=2$ jobs of the batch $\{C_1, C_2\}$, and hence jobs C_1 and C_2 are assigned to servers 2 and 3.

We note that the MDS-Reservation(0) queue, when $n=k$, is identical to a split-merge queue [20].

2) *Analysis:* Observe that under the specific scheduling policy of MDS-Reservation(0), a batch that is waiting in the buffer must necessarily have all its k jobs in the buffer, and furthermore, these k jobs go into the servers at the same time.

We now describe the Markovian representation of the MDS-Reservation(0) queue. We show that it suffices to keep track of only the total number of jobs m in the entire system.

Theorem 1: A Markovian representation of the MDS-Reservation(0) queue has a state space $\{0, 1, \dots, \infty\}$, and any state $m \in \{0, 1, \dots, \infty\}$ has transitions to: (i) state $(m+k)$ at rate λ , (ii) if $m \leq n$ then to state $(m-1)$ at rate $m\mu$, and (iii) if $m > n$ then to state $(m-1)$ at rate $(n - (n-m) \bmod k)\mu$. The MDS-Reservation(0) queue is thus a QBD process, with boundary states $\{0, 1, \dots, n-k\}$, and levels $m \in \{n-k+1+jk, \dots, n+jk\}$ for

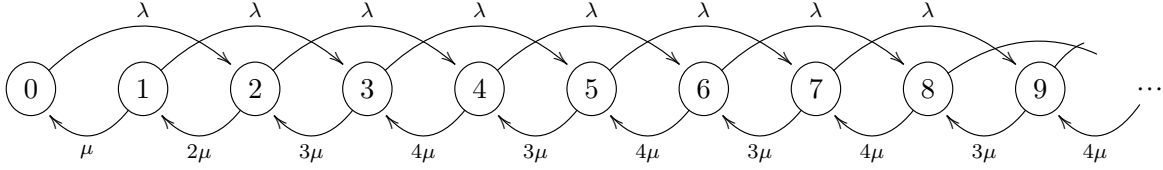


Fig. 5: State transition diagram of the MDS-Reservation(0) queue for $n=4$ and $k=2$. The notation at any state is the number of jobs m in the system in that state. The set of boundary states are $\{0,1,2\}$, and the levels are pairs of states $\{3,4\}$, $\{5,6\}$, $\{7,8\}$, etc. The transition matrix is of the form (1) with $B_0 = [0 \ 0 \ 3\mu; 0 \ 0 \ 0]$, $B_1 = [-\lambda \ 0 \ \lambda; \mu \ -(\mu+\lambda) \ 0; 0 \ 2\mu \ -(2\mu+\lambda)]$, $B_2 = [0 \ 0; \lambda \ 0; 0 \ \lambda]$, $A_0 = [0 \ 3\mu; 0 \ 0]$, $A_1 = [-(3\mu+\lambda) \ 0; 4\mu \ -(4\mu+\lambda)]$, $A_2 = [\lambda \ 0; 0 \ \lambda]$.

$$j = \{0, 1, \dots, \infty\}.$$

The state transition diagram of the MDS-Reservation(0) queue for $(n=4, k=2)$ is depicted in Fig. 5.

Theorem 1 shows that the MDS-Reservation(0) queue is a QBD process, allowing us to employ the SMC solver to obtain its steady-state distribution. Alternatively, this queue is simple enough to analyse analytically as well. Let $y(m)$ denote the number of jobs being served when the Markov chain is in state m . From the description above, this function can be written as:

$$y(m) = \begin{cases} m, & \text{if } 0 \leq m \leq n \\ n - ((n-m) \bmod k), & \text{if } m > n. \end{cases}$$

Let $\pi = [\pi_0 \ \pi_1 \ \pi_2 \ \dots]$ denote the steady-state distribution of this chain. The global balance equation for the cut between states $(m-1)$ and m gives:

$$\pi_m = \frac{\lambda}{y(m)\mu} \left(\sum_{j=(m-k)^+}^{m-1} \pi_j \right) \quad \forall m > 0. \quad (2)$$

Using these recurrence equations, for any given (n, k) , the distribution π of the number of jobs in steady-state can be computed easily.

B. MDS-Reservation(1)

1) *Scheduling policy*: The MDS-Reservation(0) scheduling policy discussed above allows the batches in the buffer to move ahead only when all k jobs in the batch can move together. The MDS-Reservation(1) scheduling policy relaxes this restriction for (only) the job at the head of the buffer. This is formalized in Algorithm 3.

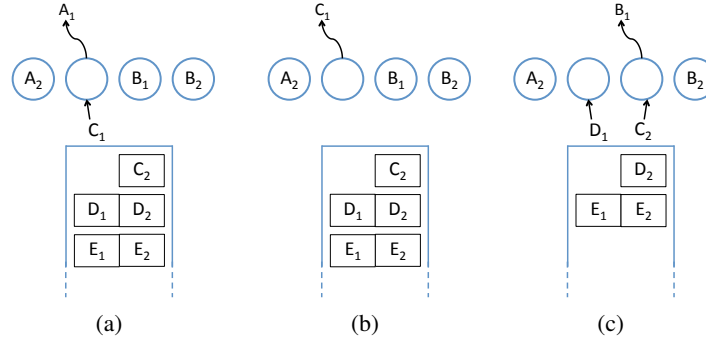


Fig. 6: An illustration of the MDS-Reservation(1) scheduling policy, for a system with parameters $(n=4, k=2)$. As shown in the figure, this policy prohibits the servers from processing jobs of the second or later batches (e.g., $\{D_1, D_2\}$ and E_1, E_2 in (b)), until they move to the top of the buffer (e.g., $\{D_1, D_2\}$ in (c)).

Algorithm 3 MDS-Reservation(1) Scheduling Policy

```

on arrival of a batch
  if buffer is empty then
    assign one job each from new batch to (at most  $k$ ) idle servers
  end if
  append remaining jobs of batch to the end of the buffer
on departure from server (say, server  $s$ ):
  if buffer is non-empty and no job from first waiting batch has been served by  $s$  then
    assign a job from first waiting batch to  $s$ 
    if first waiting batch had only one job in buffer & there exists another waiting batch then
      to every remaining idle server, assign a job from second waiting batch
    end if
  end if

```

Example 3: Consider the MDS($n=4, k=2$) queue in the state depicted in Fig. 3a. Suppose server 2 completes processing job A_1 (Fig. 6a). Under MDS-Reservation(1), server 2 now begins service of job C_1 (which is allowed by MDS, but was prohibited under MDS-Reservation(0)). Now suppose that server 2 finishes this service before any other server (Fig. 6b). In this situation, since server 2 has already processed one job from batch $\{C_1, C_2\}$, it is not allowed to process C_2 . However, there exists another batch $\{D_1, D_2\}$ in the buffer such that none of the jobs in this batch have been processed by the idle server 2. While the MDS scheduling policy would have allowed server 2 to start processing D_1 or D_2 , this is not permitted under MDS-Reservation(1), and the second server remains idle. Now, if server 3 completes service (Fig. 6c), then C_2 is assigned to server 3, allowing batch $\{D_1, D_2\}$ to move up as the first batch. This now permits server 2 to begin service of job D_1 .

The MDS-Reservation(1) scheduling policy is identical to the block-one-Scheduling policy proposed in [7]. While this scheme was analysed in [7] only for the case of $k=2$, in this paper, we present a more general analysis that holds for all values of the parameter k .

2) *Analysis:* The following theorem describes the Markovian representation of the MDS-Reservation(1) queue. Each state in this representation is defined by two quantities: (i) the total number of jobs m in the system, and (ii) the number of jobs w_1 of the first waiting batch, that are still in the buffer.

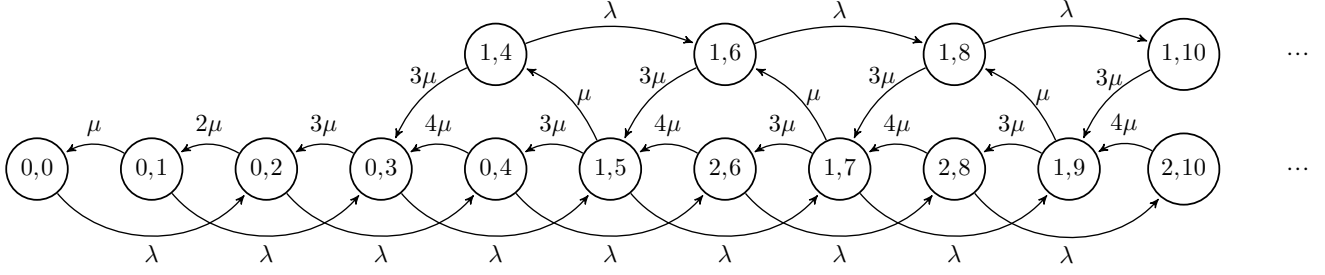


Fig. 7: State transition diagram of the MDS-Reservation(1) queue for $n=4$ and $k=2$. The notation at any state is (w_1, m) . The subset of states that are never visited are not shown. The set of boundary states are $\{0,1,2,3,4\} \times \{0,1,2\}$, and the levels are sets $\{5,6\} \times \{0,1,2\}$, $\{7,8\} \times \{0,1,2\}$, etc.

Theorem 2: The Markovian representation of the MDS-Reservation(1) queue has a state space $\{0,1,\dots,k\} \times \{0,1,\dots,\infty\}$. It is a QBD process with boundary states $\{0,\dots,k\} \times \{0,\dots,n\}$, and levels $\{0,\dots,k\} \times \{n-k+1+jk, \dots, n+jk\}$ for $j = \{1,2,\dots,\infty\}$.

The state transition diagram of the MDS-Reservation(1) queue for $(n=4, k=2)$ is depicted in Fig. 7.

Note that the state space $\{0,1,\dots,k\} \times \{0,1,\dots,\infty\}$ has several states that will never be visited during the execution of the Markov chain. For instance, the states $(w_1 > 0, m \leq n-k)$ never occur. This is because $w_1 > 0$ implies existence of some job waiting in the buffer, while $m \leq n-k$ implies that k or more servers are idle. The latter condition implies that there exists at least one idle server that can process a job from the first waiting batch, and hence the value of w_1 must be smaller than that associated to that state, thus proving the impossibility of the system being in that state.

C. MDS-Reservation(t) for a general t

1) *Scheduling policy:* Algorithm 4 formally describes the MDS-Reservation(t) scheduling policy.

Algorithm 4 MDS-Reservation(t)

```

on arrival of a batch:
  if  $\exists < t$  batches in the buffer then
    assign 1 job each from new batch to every idle server
  end if
  append remaining jobs of batch to the end of the buffer
on departure from server (say, server  $s$ ):
  find  $\hat{i} = \min\{i \geq 1 : \text{no job of the } i^{\text{th}} \text{ waiting batch has been served by } s\}$ 
  if  $\hat{i}$  exists &  $\hat{i} \leq t$  then
    assign a job of  $\hat{i}^{\text{th}}$  waiting batch to  $s$ 
    if  $\hat{i} = 1$  & the first waiting batch had only one job in buffer & there exists another waiting batch then
      to every remaining idle server, assign a job from second waiting batch
    end if
  end if

```

The following example illustrates the MDS-Reservation(t) scheduling policy when $t=2$.

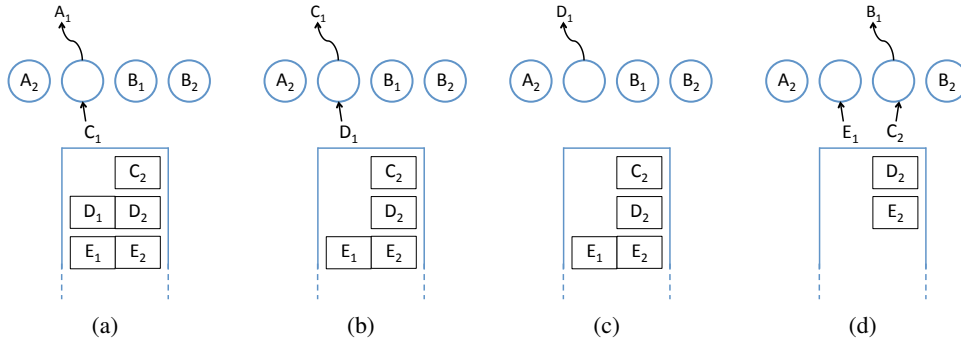


Fig. 8: An illustration of the working of the MDS-Reservation(2) scheduling policy, for a system with parameters $(n=4, k=2)$. As shown in the figure, this policy prohibits the servers from processing jobs of the third and later batches (e.g., batch $\{E_1, E_2\}$ in (c)), until they move higher in the buffer (e.g., as in (d)).

Example 4: ($t=2$). Consider the MDS($n=4, k=2$) queue in the state depicted in Fig. 3a. Suppose the second server completes processing job A_1 (Fig. 8a). Under the MDS-Reservation(2) scheduling policy, server 2 now begins service of job C_1 . Now suppose that server 2 finishes this service as well, before any other server completes its respective service (Fig. 8b). In this situation, while MDS-Reservation(1) would have mandated server 2 to remain idle, MDS-Reservation(2) allows it to start processing a job from the next batch $\{D_1, D_2\}$. However, if the server also completes processing of this job before any other server (Fig. 8c), then it is not allowed to take up a job of the third batch $\{E_1, E_2\}$. Now suppose server 3 completes service (Fig. 8d). Server 3 can begin serving job C_2 , thus clearing batch $\{C_1, C_2\}$ from the buffer, and moving the two remaining batches up in the buffer. Batch $\{E_1, E_2\}$ is now within the threshold of $t=2$, allowing it to be served by the idle server 2.

2) Analysis:

Theorem 3: The Markovian representation of the MDS-Reservation(t) queue has a state space $\{0, 1, \dots, k\}^t \times \{0, 1, \dots, \infty\}$. It is a QBD process with boundary states $\{0, \dots, k\}^t \times \{0, \dots, n - k + tk\}$, and levels $\{0, \dots, k\}^t \times \{n - k + 1 + jk, \dots, n + jk\}$ for $j = \{t, t+1, \dots, \infty\}$.

One can see that the sequence of MDS-Reservation(t) queues, as t increases, becomes closer to the MDS queue. This results in tighter bounds, and also increased complexity of the transition diagrams. The limit of this sequence is the MDS queue itself.

Theorem 4: The MDS-Reservation(t) queue, when $t = \infty$, is precisely the MDS queue.

V. THE $M^k/M/n(t)$ QUEUES: UPPER BOUNDS

In this section, we present a class of scheduling policies (and resulting queues), which we call the $M^k/M/n(t)$ scheduling policies (and $M^k/M/n(t)$ queues), whose performance upper bounds the performance of the MDS queue. The scheduling policies presented here relax the constraint of requiring the k jobs in a batch to be processed by k distinct servers. While the $M^k/M/n(t)$ scheduling policies and the $M^k/M/n(t)$ queues are not realizable in practice, they are presented here only to obtain upper bounds to the performance of the MDS queue.

The $M^k/M/n(t)$ scheduling policy, in a nutshell, is as follows:

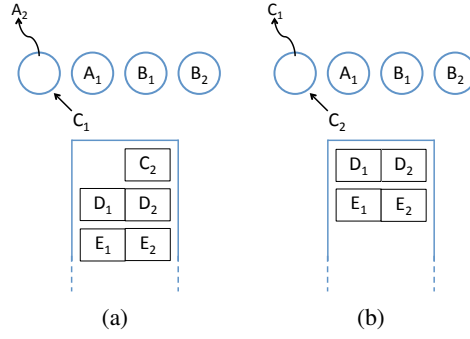


Fig. 9: Illustration of the working of the $M^k/M/n(0)$ scheduling policy. This policy allows a server to process more than one jobs of the same batch. As shown in the figure, server 1 processes both C_1 and C_2 .

“apply the MDS scheduling policy whenever there are t or fewer batches in the buffer; when there are more than t batches in the buffer, ignore the restriction requiring the k jobs of a batch to be processed by distinct servers.”

We first describe the $M^k/M/n(0)$ queue in detail, before moving on to the general $M^k/M/n(t)$ queues.

A. $M^k/M/n(0)$

1) *Scheduling policy:* The $M^k/M/n(0)$ scheduling policy operates by completely ignoring the restriction of assigning distinct servers to jobs of the same batch. This is described formally in Algorithm 5.

Algorithm 5 $M^k/M/n(0)$

```

on arrival of a batch
  assign jobs of this batch to idle servers (if any)
  append remaining jobs at the end of the buffer
on departure from a server
  if buffer is not empty then
    assign a job from the first waiting batch to this server
  end if

```

Note that the $M^k/M/n(0)$ queue is identical to the $M^k/M/n$ queue, i.e., an $M/M/n$ queue with batch arrivals.

The following example illustrates the working of the $M^k/M/n(0)$ scheduling policy.

Example 5: Consider the MDS($n=4, k=2$) queue in the state depicted in Fig. 3a. Suppose the first server completes processing job A_2 , as shown in Fig. 9a. Under the $M^k/M/n(0)$ scheduling policy, server 1 now takes up job C_1 . Next suppose server 1 also finishes this task before any other server completes service (Fig. 9b). In this situation, the MDS scheduling policy would prohibit job C_2 to be served by the first server. However, under the scheduling policy of $M^k/M/n(0)$, we relax this restriction, and permit server 1 to begin processing C_2 .

2) *Analysis:* We now describe a Markovian representation of the $M^k/M/n(0)$ scheduling policy, and show that it suffices to keep track of only the total number of jobs m in the system.

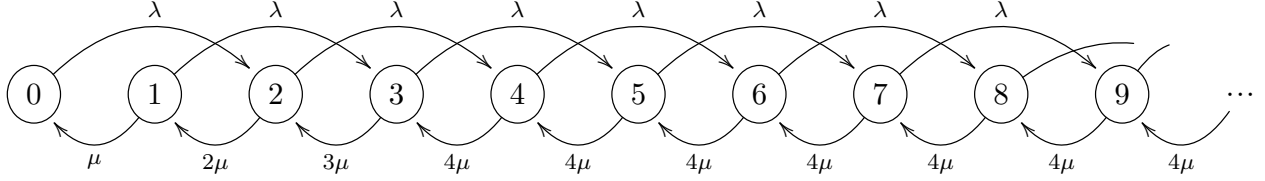


Fig. 10: State transition diagram of the $M^k/M/n(0)$ queue for $n=4$ and $k=2$. The notation at any state is the number of jobs m in the system in that state. The set of boundary states are $\{0,1,2,3,4\}$, and the levels are pairs of states $\{5,6\}$, $\{7,8\}$, etc. The transition matrix is of the form (1) with $B_0 = [0 \ 0 \ 0 \ 0 \ 4\mu; 0 \ 0 \ 0 \ 0 \ 0]$, $B_1 = [-\lambda \ 0 \ \lambda \ 0 \ 0; \mu \ -(\mu+\lambda) \ 0 \ \lambda \ 0; 0 \ 2\mu \ -(2\mu+\lambda) \ 0 \ \lambda; 0 \ 0 \ 3\mu \ -(3\mu+\lambda) \ 0; 0 \ 0 \ 0 \ 4\mu \ -(4\mu+\lambda)]$, $B_2 = [0 \ 0; 0 \ 0; 0 \ 0; \lambda \ 0; 0 \ \lambda]$, $A_0 = [0 \ 4\mu; 0 \ 0]$, $A_1 = [-(4\mu+\lambda) \ 0; 4\mu \ -(4\mu+\lambda)]$, $A_2 = [\lambda \ 0; 0 \ \lambda]$.

Theorem 5: The Markovian representation of the $M^k/M/n(0)$ queue has a state space $\{0,1,\dots,\infty\}$, and any state $m \in \{0,1,\dots,\infty\}$, has transitions (i) to state $(m+k)$ at rate λ , and (ii) if $m > 0$, then to state $(m-1)$ at rate $\min(n,m)\mu$. It is a QBD process with boundary states $\{0,\dots,k\} \times \{0,\dots,n\}$, and levels $\{0,\dots,k\} \times \{n-k+1+jk,\dots,n+jk\}$ for $j = \{1,2,\dots,\infty\}$.

The state transition diagram of the $M^k/M/n(0)$ queue for $n=4, k=2$ is shown in Fig. 10.

Theorem 5 shows that the MDS-Reservation(0) queue is a QBD process, allowing us to employ the SMC solver to obtain its steady-state distribution. Alternatively, this queue is simple enough to analyse analytically as well. Let π_m denote the stationary probability of any state $m \in \{0,1,\dots,\infty\}$. Then, for any $m \in \{1,\dots,\infty\}$, the global balance equation for the cut between states $(m-1)$ and m gives:

$$\pi_m = \frac{\lambda}{\min(m,n)\mu} \sum_{j=(m-k)^+}^{m-1} \pi_j. \quad (3)$$

The stationary distribution of the Markov chain can now be computed easily from these equations.

B. $M^k/M/n(t)$ for a general t

1) *Scheduling policy:* Algorithm 6 formally describes the $M^k/M/n(t)$ scheduling policy.

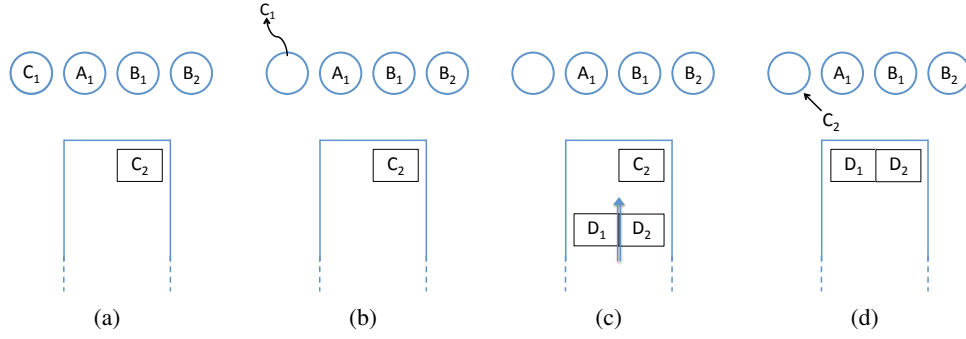


Fig. 11: Illustration of the working of the $M^k/M/n(1)$ scheduling policy. This policy allows a server to begin processing a job of a batch that it has already served, unless this batch is the only batch waiting in the buffer. As shown in the figure, server 1 cannot process C_2 in (b) since it has already processed C_1 and C is the only waiting batch; this restriction is removed upon arrival of another batch in the buffer in (d).

Algorithm 6 $M^k/M/n(t)$

```

on arrival of a batch
  if buffer has strictly fewer than  $t$  batches then
    assign jobs of new batch to idle servers
  else if buffer has  $t$  batches then
    assign jobs of first batch to idle servers
    if first batch is cleared then
      assign jobs of new batch to idle servers
    end if
  end if
  append remaining jobs of the new batch to the end of the buffer
on departure of job from a server (say, server  $s$ )
  if number of batches in buffer is strictly greater than  $t$  then
    assign job from first batch in buffer to this server
  else
    among all batches in the buffer that have not been served by  $s$ , find the one that arrived earliest
    assign a job of this batch to  $s$ 
  end if

```

Example 6: ($t=1$). Consider a system in the state shown in Fig. 11a. Suppose server 1 completes execution of job C_1 (Fig. 11b). In this situation, the processing of C_2 by server 1 would be allowed under $M^k/M/n(0)$, but prohibited in the MDS queue. The $M^k/M/n(1)$ queue follows the scheduling policy of the MDS queue whenever the total number of batches in the buffer is no more than 1, and hence in this case, server 1 remains idle. Next, suppose there is an arrival of a new batch (Fig. 11c). At this point there are two batches in the buffer, and the $M^k/M/n(1)$ scheduling policy switches its mode of operation to allowing any server to serve any job. Thus, the first server now begins service of C_2 (Fig. 11d).

2) *Analysis:*

Theorem 6: The state transition diagram of the $M^k/M/n(t)$ queue has a state space $\{0, 1, \dots, k\}^t \times \{0, 1, 2, \dots\}$. It

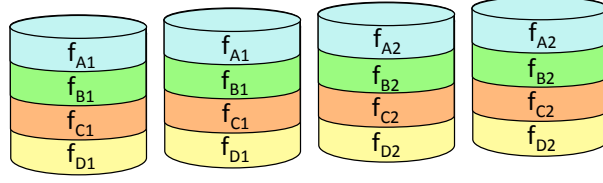


Fig. 12: An example of the Replication-II storage scheme. Here, $n = 4$ and $k = 2$. Contrast this to the Replication-I storage scheme depicted in Fig. 1a for the same parameters.

is a QBD process with boundary states $\{0, \dots, k\}^t \times \{0, \dots, n + tk\}$, and levels $\{0, \dots, k\}^t \times \{n - k + 1 + jk, \dots, n + jk\}$ for $j = \{t + 1, t + 2, \dots, \infty\}$.

As in the case of MDS-Reservation(t) queues, one can see that the sequence of $M^k/M/n(t)$ queues, as t increases, becomes closer to the MDS queue. On increase in the value of parameter t , the bounds become tighter, but the complexity of the transition diagrams also increases, and the limit of this sequence is the MDS queue itself.

Theorem 7: The $M^k/M/n(t)$ queue, when $t = \infty$, is precisely the MDS(n, k) queue.

Remark 1: The class of queues presented in this section have another interesting intellectual connection with the MDS queue: the performance of an MDS(n, k) queue is lower bounded by the $M^k/M/(n-k+1)(t)$ queue for any value of t .

VI. PERFORMANCE ANALYSIS AND COMPARISON OF VARIOUS QUEUES

In this section we analyse the performance of the MDS-Reservation(t) queues and the $M^k/M/n(t)$ queues using the properties of QBD processes. These form lower and upper bounds respectively to the performance of the MDS queue. We also provide a performance analysis of the MDS queue via MCMC simulations. We compare the performance of these queues with replication-based schemes.

A replication-based scheme stores multiple copies of the entire data, and hence must have n as a multiple of k . As in systems employing MDS codes, we assume that under a replication-based scheme, each server has a capacity of storing $\frac{1}{k}$ of the total data. Replication can be of two types:

- Replication-I (depicted in Fig. 1a): The n servers are partitioned into a k sets of $\frac{n}{k}$ servers each, and the set of files are partitioned into k sets of equal size. Each set of files is associated to one set of servers, and is replicated in the $\frac{n}{k}$ servers in this set. Thus, a request for reading a file can be completed by any of the $\frac{n}{k}$ servers in the set of servers to which the file belongs.
- Replication-II (depicted in Fig. 12): The n servers are partitioned into a k sets of $\frac{n}{k}$ servers each, and each individual file is split into k chunks of equal size. For each file, the i^{th} chunk ($1 \leq i \leq k$) is replicated in the $\frac{n}{k}$ servers of the i^{th} set of servers. Thus, a request for reading a file is split into k identical jobs, and the i^{th} job can be served by any of the $\frac{n}{k}$ servers in the i^{th} set of servers.

In our study, we observed that Replication-I consistently performs poorer than Replication-II, under the models considered here. Thus, in the analysis to follow, we plot only the performance of Replication-II scheme, and omit Replication-I. Furthermore, in the analysis of the latency incurred by coded systems, we assume that the additional delay incurred due to computations required for decoding the data is negligible. This assumption is justified in most systems of interest, since the codes employed typically have small block lengths, allowing for fast decoding.

The following subsections present the analysis and comparisons of the queues with respect to various metrics. Unless mentioned otherwise, the values of the system parameters for the graphs plotted are

$$n=10, k=5, \text{ and } \mu=1 .$$

The system is assumed to be in steady-state. The curves marked '[MCMC]' have been obtained via MCMC simulations, while the rest have been obtained analytically.

A. Maximum throughput

The maximum throughput is the maximum possible number of requests that can be served by the system per unit time.

Theorem 8: For any given (n, k) and $t \geq 1$, let $\lambda_{\text{Resv}(t)}^*$, λ_{MDS}^* , and $\lambda_{M^k/M/n(t)}^*$ denote the maximum throughputs of the MDS-Reservation(t), MDS, and $M^k/M/n(t)$ queues respectively. Then,

$$\lambda_{\text{MDS}}^* = \lambda_{M^k/M/n(t)}^* = \frac{n}{k}\mu . \quad (4)$$

When k is treated as a constant,

$$(1 - O(n^{-2})) \frac{n}{k}\mu \leq \lambda_{\text{Resv}(t)}^* \leq \frac{n}{k}\mu . \quad (5)$$

In particular, when $k = 2$,

$$\left(1 - \frac{1}{2n^2 - 2n + 1}\right) \frac{n}{k}\mu = \lambda_{\text{Resv}(1)}^* \leq \lambda_{\text{Resv}(t)}^* , \quad (6)$$

and when $k = 3$,

$$\left(1 - \frac{4n^3 - 8n^2 + 2n + 4}{3n^5 - 12n^4 + 22n^3 - 29n^2 + 26n - 8}\right) \frac{n}{k}\mu = \lambda_{\text{Resv}(1)}^* \leq \lambda_{\text{Resv}(t)}^* . \quad (7)$$

The maximum throughput of Replication-I and Replication-II is also $\frac{n}{k}\mu$.

Note that the special case of Theorem 8, for the MDS-Reservation(1) queue with $k=2$, also recovers the throughput results of [7]. Moreover, as compared to the proof techniques presented in [7], the proofs in this paper are simpler and do not require computation of the stationary distribution. Using the techniques presented in the proof of Theorem 8, bounds analogous to (6) and (7) can be computed for $k \geq 4$ as well.

Fig. 13 plots loss in maximum throughput incurred by the MDS-Reservation(1) and the MDS-Reservation(2) queues, as compared to that of the MDS queue (and replication).

B. System occupancy

The system occupancy at any given time is the number of jobs present in the system at that time. This includes jobs that are waiting in the buffer as well as the jobs being processed in the servers at that time. Fig. 14 plots the complementary cdf of the the number of jobs in the system in the steady state. Observe that the analytical upper and lower bounds of the MDS-Reservation(2) and the $M^k/M/n(1)$ queues respectively are extremely close to each other.

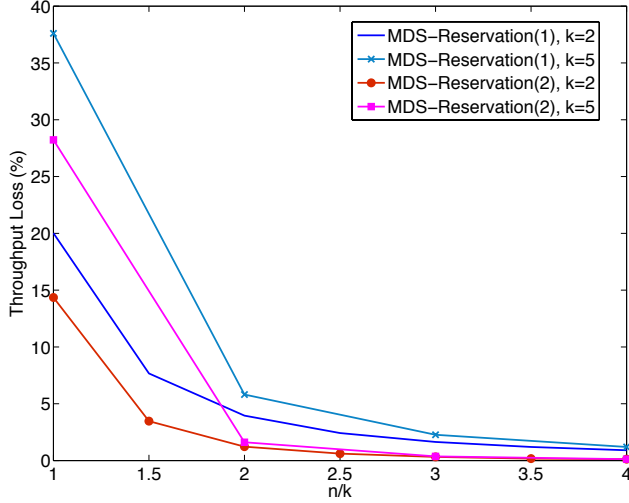


Fig. 13: Loss in maximum throughput incurred by the MDS-Reservation(1) and the MDS-Reservation(2) queues as compared to that of the MDS queue (and replication).

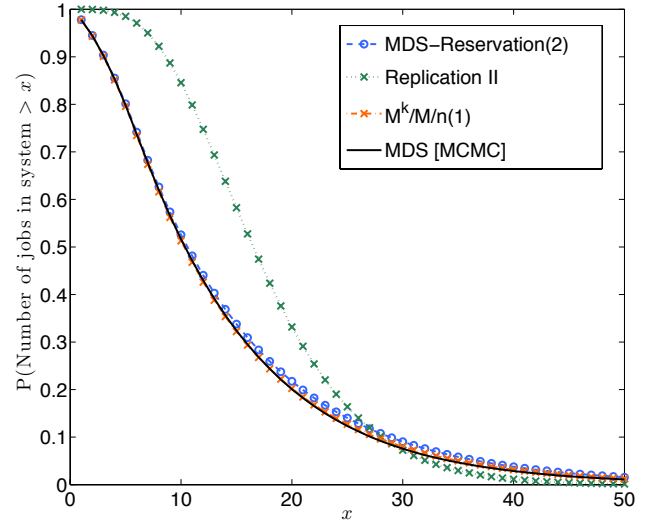


Fig. 14: Steady state distribution (complementary cdf) of the system occupancy.

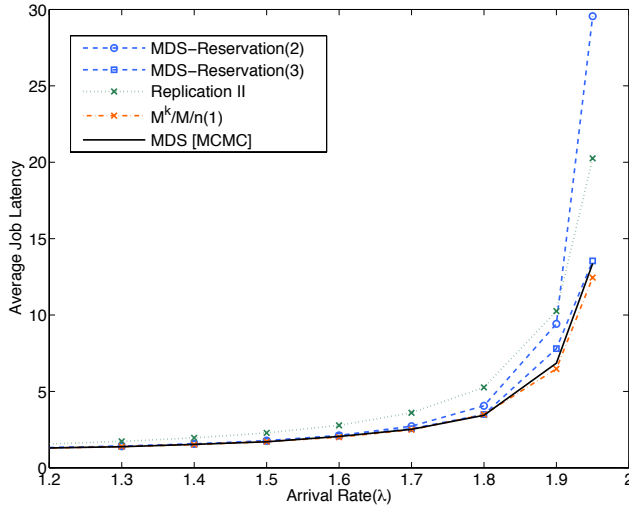


Fig. 15: Average latency faced by a job.

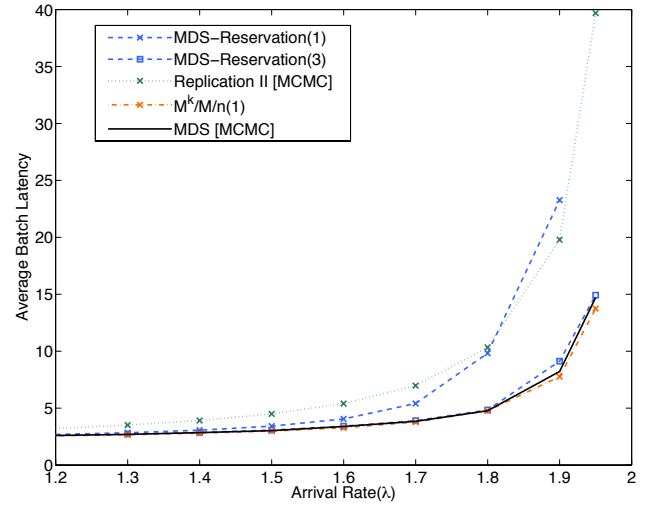


Fig. 16: Average latency faced by a batch.

C. Average job latency

The latency faced by a job is the time from its arrival into the system till the time it completes being serviced at a server. Fig. 15 plots the average latency faced by a job in the steady state. Here, the average job latencies of the MDS-Reservation(t) and the M^k/M/n(t) queues have been obtained analytically by finding the stationary distribution of the respective QBD processes, and applying Little's law. The average job latency of the MDS queue is plotted via an MCMC simulation.

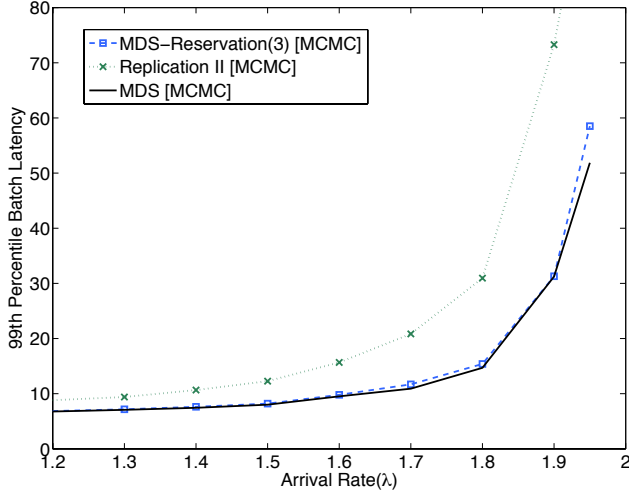


Fig. 17: 99th percentile of the distribution of the latency faced by a batch.

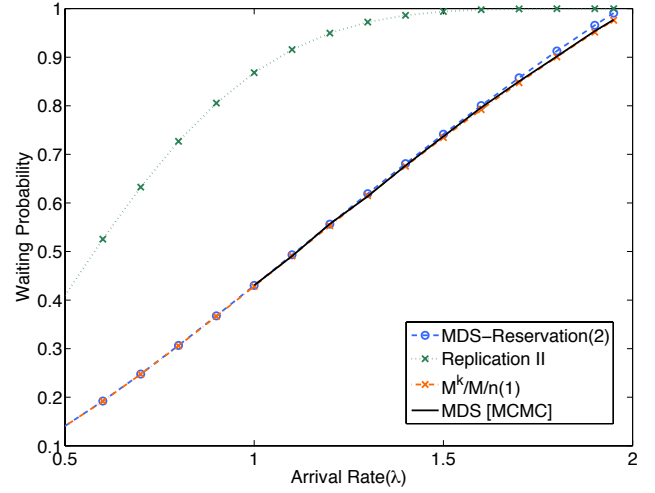


Fig. 18: Waiting probability.

D. Batch latency

The latency faced by a batch is the time from its arrival into the system till the last of its k jobs completes service.

1) *Average*: We have analytically computed the average batch latencies of the MDS-Reservation(t) and the $M^k/M/n(t)$ queues in the following manner. We first compute the steady state distribution π of the corresponding QBD process. As discussed in the previous sections, each state i is associated to a unique configuration of the jobs in the system. Now, the average latency d_i faced by a batch entering when the system is in state i can be computed easily as a dynamic program, by employing the Markovian representations of the queues presented previously. Since Poisson arrivals see time averages [21], the average latency faced by a batch in the steady state is given by $\sum_i \pi_i d_i$.

Fig. 16 plots the average latency faced by a batch in the steady state. Observe that the coded systems achieve upto 70% reduction in average latency as compared to replication. Also observe that the performance of the MDS-Reservation(t) scheduling policy, for t as small as 3, is extremely close to that of the MDS scheduling policy and to the upper bounding $M^k/M/n(1)$ scheduling policy.

2) *Tails*: It is of considerable practical interest to analyse the tails distribution of the latency. Here, we perform this analysis via MCMC simulations. Fig. 17 plots the 99th percentile of the distribution of the latency faced by a batch: for a given value of the arrival rate λ , a curve takes value y_λ if exactly 1% of the batches incur a delay greater than y_λ . Again, we see that coded systems achieve significant gains over replication.

E. Waiting probability

The waiting probability is the probability that, in the steady state, one or more jobs of an arriving batch will have to wait in the buffer and will not be served immediately. Fig. 18 plots the waiting probability for the different queues considered in the paper. Observe how tightly the $M^k/M/n(1)$ and the MDS-Reservation(2) queues bound the waiting probability of the MDS queue. Also observe that replication fares much worse than codes, even for low arrival rates.

F. Blocking Probability

We have assumed throughout the paper that the buffer has an infinite capacity. In this section, let us suppose that the buffer has a finite capacity of accommodating at most a certain number batches. Then, the blocking probability is the probability that (in the steady state) an arriving batch will find the buffer full and hence will have to be rejected. Fig. 19 plots the blocking probability of a system for various buffer sizes. Observe that the blocking probability of the coded system is consistently smaller than that of replication.

G. Degraded Reads

The coded system presented so far assumes that each incoming request desires one complete file F_x , for some $x \in \{A, B, C, \dots\}$. As discussed previously in Section II-A4, in certain applications, incoming requests may sometimes require only a part of the file $f_{x\ell}$, for some $\ell \in \{1, \dots, k\}$. In this case, assuming the code is systematic, $f_{x\ell}$ can be retrieved by reading it directly from the ℓ^{th} server. However, in the situation that server ℓ is busy, one may need to perform a ‘degraded read’, i.e., recover $f_{x\ell}$ from the data stored in the remaining $(n-1)$ servers.

Under any MDS code, a degraded read may be performed by reconstructing the entire file F_x from the data stored in any k of the $(n-1)$ remaining servers, and subsequently extracting the desired chunk $f_{x\ell}$ from it. An alternative means of performing degraded reads in MDS codes has been proposed recently in [12]–[14]. In particular, practical codes, termed the ‘product-matrix (PM) codes’, have been proposed in [14], that can potentially provide significant speed ups to the degraded read operation. These codes are associated to an additional parameter d , and can recover $f_{x\ell}$ of file F_x by reading and downloading a fraction $\frac{1}{d-k+1}^{\text{th}}$ of the data stored in *any* d of the remaining $(n-1)$ servers. For example, consider a system with parameters $n=6$ and $k=2$, that stores files encoded by a PM code with parameter $d=3$. Then, a complete file can be recovered, as in any other MDS code, by downloading the corresponding chunks from any $k=2$ servers. A part $f_{x\ell}$ of a file may also be recovered by reading and downloading data that is half the size of $f_{x\ell}$ from each of any $d=3$ servers. This method of performing a degraded read is termed as a ‘repair’ operation, since it was first proposed in the context of repairing failed servers.

In this paper, we employ the framework of the MDS queue developed in this paper, to compare the two aforementioned methods of performing degraded reads. We assume that each request desires to read chunk $f_{x\ell}$ for some (uniformly) random $x \in \{A, B, \dots\}$ and $\ell \in \{1, \dots, k\}$. We further assume the time required to read data from any server to be exponentially distributed with a mean equal to the amount of data to be read, with each chunk $f_{x\ell}$ assumed to be of unit size. Under this setting, the method of reconstructing F_x from any k servers forms an MDS($n-1, k$) queue (with $\mu=1$), and the method of performing ‘repair’ forms an MDS($n-1, d$) queue (with $\mu = \frac{1}{d-k+1}$).

In Fig. 20, we plot the average latency incurred under these two methods of performing degraded reads, for the parameters $n=6$, $k=2$ and $d=3$. One can see that the method of ‘repair’ performs consistently better as compared to reconstructing the entire file F_x , thus corroborating the efficacy of these codes in terms of latency performance for degraded reads.

H. Request-flooding

Request-flooding [9]–[11] is a method of (possibly) reducing the latency by sending a request to more than the minimum requisite number of servers. Under an (n, k) MDS code, a request-flooding strategy sends (redundant)

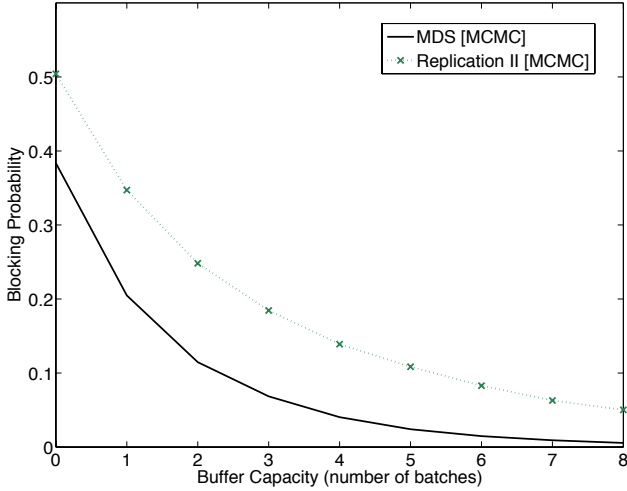


Fig. 19: Blocking probability when the buffer has a finite capacity. The arrival rate is $\lambda = 1.5$.

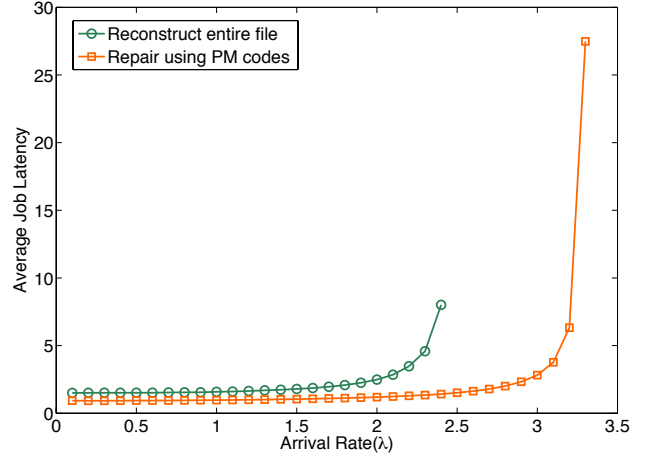


Fig. 20: Average latency during degraded reads. The parameters associated to this system are $n = 6$, $k = 2$. The service time is exponentially distributed with a mean proportional to the amount of data to be read. The parameter d associated to the product-matrix (PM) codes is set as $d = 3$.

requests to some ρ ($k \leq \rho \leq n$) servers. Upon completion of service by any k of these ρ servers, the batch is considered to be served, and the services of the remaining $(\rho - k)$ jobs of that batch are cancelled. The case of $\rho = k$ corresponds to the absence of flooding, and corresponds to the case considered throughout the rest of this paper. We term ρ as the *request-flooding parameter*. Request-flooding in a coded storage system was previously analysed in [11] for the case $\rho = n$ (in the absence of cancellation costs).

In this section, via MCMC models, we analyse and compare the performance of request-flooding to a ρ number of servers. As discussed previously in Section II-A3, the cancellation of jobs would typically mandate a certain cost. In our simulations, we account for this in the following manner. Upon cancellation of a job at a server, we require the server to not accept any further jobs (and remain idle) for a time duration that is distributed exponentially with a rate 10μ (loosely speaking, this amounts to a 10% cancellation cost). In Fig. 21, we compare the average batch latency incurred for various values of the flooding parameter $\rho \in \{k = 5, 6, 7, 8, 9, n = 10\}$ under such a flooding scheme. Observe that when the rate of arrivals λ is low, the average latency is minimized when requests are flooded to all $n = 10$ servers. However, as the arrival rate λ increases, the benefits of flooding diminish, and at very high arrival rates it seems prudent to flood only to a small number of servers, or to not flood at all.

The intuition behind the plots of Fig. 21 is as follows. Firstly, observe that flooding a batch to a larger number of servers ensures faster service for that batch. However, this also mandates additional resource utilization due to flooding and cancellation. The performance of flooding schemes at different arrival rates depends on the magnitude of the adverse effects of this additional utilization of resources. For instance, when the buffer is empty, the additional resource utilization due to flooding and cancellation does not affect the performance, since there is no other batch waiting for these resources. This intuitive argument explains the superior performance of flooding with a high ρ at low arrival rates. On the other hand, when the buffer has several batches waiting in it, the additional resource utilization by one batch prevents the waiting batches from utilizing these resources during

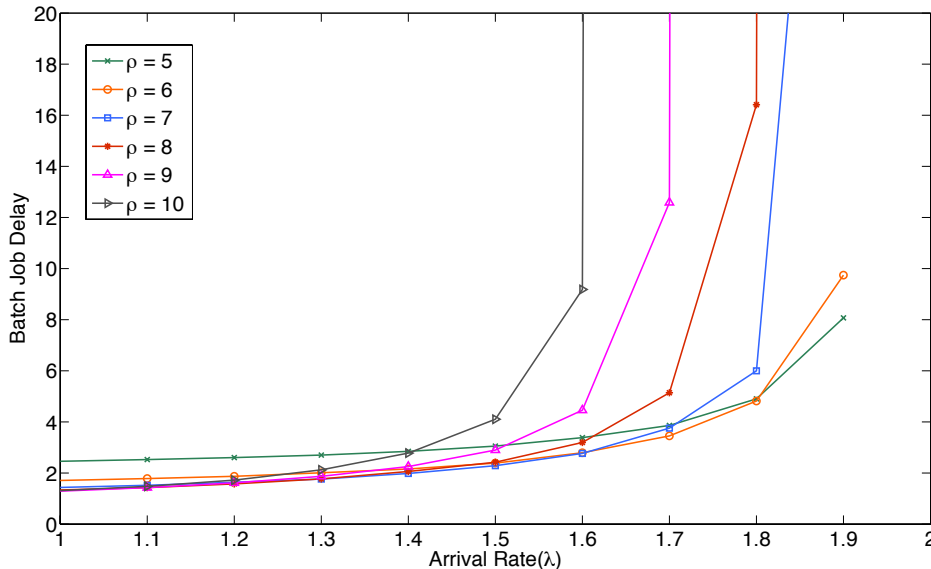


Fig. 21: Average batch latency with request-flooding (MCMC simulations). ρ is the flooding parameter. Cancelling a job at a server incurs a cancellation cost: the server cannot take up another job for a time duration that is exponentially distributed with a rate 10μ .

that time. This increase in the waiting times dominates the gains achieved by a faster service, and hence we observe a superior performance of smaller values of ρ when the arrival rate is high. We note that in any system, in general, such a crossover may occur, with the crossover point being dependent on the cost of cancellation.

VII. DISCUSSION

In this paper, we study storage systems that employ Maximum-Distance-Separable(MDS) codes through the lens of queueing theory. The queues under such systems are termed the MDS queues, and an analysis of their latency performance is provided through analytical means as well as via MCMC-based simulations. Our analysis reveals the superior performance of codes compared to that of currently used replication-based schemes (upto 70%) under the models considered here. The key insight is that, the property of being able to recover the data from *any* k of the servers that lends MDS codes a high storage efficiency and reliability, is also instrumental in providing them a low latency. This paper thus makes a case for the use of codes to store frequently accessed “hot” data.

We present two classes of scheduling policies, the MDS-Reservation(t) and the $M^k/M/n(t)$ scheduling policies, that respectively lower and upper bound the MDS queue. We show that the queues resulting from these scheduling policies belong to the class of Quasi-Birth-Death (QBD) processes, and exploit the properties of the QBD processes in our analysis. While both these classes converge to the MDS queue when the parameter $t = \infty$, we show that even for small values of t (as small as $t = 3$), these bounds are quite tight and suffice to characterize the precise performance of the MDS queue. The MDS-Reservation(t) scheduling policy represents a practical scheme that requires the maintenance of a much smaller state, and may be of independent interest in practical systems.

We also use the framework of the MDS queue developed in this paper to compare different methods of performing degraded reads. We observe that the average latency incurred for a degraded read can be reduced

significantly by employing codes with efficient ‘repair’ properties, thereby providing a queueing-theoretic evidence for the efficacy of these codes. In the paper, we also study the efficacy of request flooding. We observe that when the arrival rates are low, flooding requests to a large number of servers yields a considerable improvement in the latency, while at high arrival rates it is better to flood to only a few servers or to not flood at all.

In the future, we intend to build upon the framework presented in this paper, and analyse queues that relax one or more of the assumptions made in the paper:

- General service times: more accurate modelling based on the underlying properties of the storage devices (e.g., [22]).
- Heterogeneous requests: files of different sizes.
- Heterogeneous servers: the n servers may not have identical service-time distributions.
- Absence of MDS property: a storage system may alternatively employ codes that are not MDS (e.g., [4]), or store different files in different sets of n servers. In such situations, instead of being able to recover the data from *any* set of k servers, there will exist pre-specified subsets of the servers from which a file can be recovered.
- Decentralized MDS queue: each server may have its own buffer, in which case, the k jobs of a batch must be sent to the buffers of k distinct servers, and the choice of these k servers may have to be made much before the jobs are actually serviced.

Finally, it is also of utmost interest to connect the framework and results presented in this paper to traces from real-world data centers, which we plan to do in the near future.

REFERENCES

- [1] D. Borthakur, “HDFS and Erasure Codes (HDFS-RAID),” 2009. [Online]. Available: <http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html>
- [2] “Erasure codes: the foundation of cloud storage,” Sep. 2010. [Online]. Available: <http://blog.cleversafe.com/?p=508>
- [3] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in globally distributed storage systems,” in *9th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2010.
- [4] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure coding in Windows Azure Storage,” in *USENIX Annual Technical Conference (ATC)*, Jun. 2012.
- [5] F. MacWilliams and N. Sloane, *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, 1977.
- [6] I. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.
- [7] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, “Codes can reduce queueing delay in data centers,” in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Cambridge, Jul. 2012.
- [8] U. Ferner, M. Medard, and E. Soljanin, “Toward sustainable networking: Storage area networks with network coding,” Oct. 2012.
- [9] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker, “More is less: Reducing latency via redundancy,” in *11th ACM Workshop on Hot Topics in Networks*, Oct. 2012, pp. 13–18.
- [10] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, “Why let resources idle? aggressive cloning of jobs with dolly,” in *USENIX HotCloud*, Jun. 2012.
- [11] G. Joshi, Y. Liu, and E. Soljanin, “Coding for fast content download,” in *50th Annual Allerton Conference on Communication, Control, and Computing, 2012.*, Oct. 2012.
- [12] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, “Network coding for distributed storage systems,” *IEEE Transactions on Information Theory*, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.
- [13] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran, “Explicit construction of optimal exact regenerating codes for distributed storage,” in *Proc. 47th Annual Allerton Conference on Communication, Control, and Computing*, Urbana-Champaign, Sep. 2009, pp. 1243–1249.
- [14] K. V. Rashmi, N. B. Shah, and P. V. Kumar, “Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction,” *IEEE Transactions on Information Theory*, vol. 57, no. 8, pp. 5227–5239, Aug. 2011.
- [15] S. Borkar, “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation,” *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, 2005.

- [16] F. Baccelli, A. Makowski, and A. Shwartz, "The fork-join queue and related systems with synchronization constraints: stochastic ordering and computable bounds," *Advances in Applied Probability*, pp. 629–660, 1989.
- [17] E. Varki, "Response time analysis of parallel computer and storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 11, pp. 1146–1161, 2001.
- [18] R. Feldman and C. Valdez-Flores, *Applied Probability and Stochastic Processes (Chapter 13)*. Springer, 2010.
- [19] D. Bini, B. Meini, S. Steffé, and B. Van Houdt, "Structured markov chains solver: software tools," in *Proceedings of the 2006 workshop on Tools for solving structured Markov chains*, 2006.
- [20] P. Harrison and S. Zertal, "Queueing models with maxima of service times," *Computer Performance Evaluation. Modelling Techniques and Tools*, pp. 152–168, 2003.
- [21] R. Wolff, "Poisson arrivals see time averages," *Operations Research*, vol. 30, no. 2, pp. 223–231, 1982.
- [22] A. Merchant and P. Yu, "Analytic modeling and comparisons of striping strategies for replicated disk arrays," *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 419–433, 1995.

APPENDIX

PROOFS OF THEOREMS

Proof of Theorem 1: Since the scheduling policy mandates all k jobs of any batch to start service together, the number of jobs in the buffer is necessarily a multiple of k . Furthermore, when the buffer is not empty, the number of servers that are idle must be strictly smaller than k (since otherwise, the first waiting batch can be served). It follows that when $m \leq n$, the buffer is empty ($b = 0$), and all m jobs are being served by m servers (and $z = (n - m)$ servers are idle). When $m > n$, the buffer is not empty. Assuming there are z idle servers, there must be $(n - z)$ jobs currently being served, and hence there are $m - (n - z)$ jobs in the buffer. However, since the number of jobs in the buffer must be a multiple of k , and since $z \in \{0, 1, \dots, k - 1\}$, it must be that $z = (n - m) \bmod k$. Thus, when $m > n$, there are $b = \frac{m - n + z}{k}$ batches waiting in the buffer, and $w_i = k$, $s_i = 0 \ \forall i \in \{1, \dots, b\}$. We have thus shown that the knowledge of m suffices to completely describe the system. ■

Once we have determined the configuration of the system as above, it is now easy to obtain the transitions between the states. An arrival of a batch increases the total number of jobs in the system by k , and hence the transition from state m to $(m + k)$ at rate λ . When $m \leq n$, all m jobs are being served, and the buffer is empty. Thus, the total number of jobs in the system reduces to $(m - 1)$ at rate $m\mu$. When $m > n$, the number of jobs being served is $n - z = n - ((n - m) \bmod k)$, and thus there is a transition from state m to $(m - 1)$ at rate $(n - ((n - m) \bmod k))\mu$. ■

Proof of Theorem 2: Results as a special case of Theorem 3. As a side note, in any given state $(w_1, m) \in \{0, 1, \dots, k\} \times \{0, 1, \dots, \infty\}$ of the resulting Markov chain, the number of idle servers is given by $z = n - m$ if $m \leq n - k$, and $z = (n + w_1 - m) \bmod k$ otherwise. The state (w_1, m) has transitions to state:

- $((m + k - n)^+, m + k)$ at rate λ , if $w_1 = 0$.
 - $(w_1, m + k)$ at rate λ , if $w_1 \neq 0$
 - $(w_1, m - 1)$ at rate $m\mu$, if $w_1 = 0$.
 - $(w_1, m - 1)$ at rate $(k - w_1 - z)\mu$, if $w_1 \neq 0$
 - $(w_1 - 1, m - 1)$ at rate $(n - k + w_1)\mu$, if $(w_1 > 1 \text{ or } (w_1 = 1 \ \& \ m \leq n + 1))$
 - $((k - z)^+, m - 1)$ at rate $(n - k + w_1)\mu$, if $(w_1 = 1 \ \& \ m > n + 1)$.
-

Proof of Theorem 3: For any state of the system $(w_1, w_2, \dots, w_t, m) \in \{0, 1, \dots, k\}^t \times \{0, 1, \dots, \infty\}$, define

$$q = \begin{cases} 0 & \text{if } w_1 = 0 \\ t & \text{else if } w_t \neq 0 \\ \arg \max \{t' : w_{t'} \neq 0, 1 \leq t' \leq t\} & \text{otherwise.} \end{cases} \quad (8)$$

It can be shown that

$$b = \begin{cases} 0 & \text{if } q = 0 \\ q & \text{if } 0 < q < t \\ t + \left\lfloor \frac{m - \sum_{j=1}^t w_j - n}{k} \right\rfloor & \text{otherwise,} \end{cases} \quad (9)$$

$$z = n - (m - \sum_{j=1}^t w_j - (b - t)^+ k), \quad (10)$$

$$\text{for } i \in \{1, \dots, b\}, \quad s_i = \begin{cases} w_{i+1} - w_i & \text{if } i \in \{1, \dots, q-1\} \\ k - z - w_q & \text{if } i = q \\ 0 & \text{if } i \in \{q+1, \dots, b\}, \end{cases} \quad (11)$$

and

$$\text{for } i \in \{t+1, \dots, b\}, \quad w_i = k. \quad (12)$$

Given the complete description of the state of the system as above, the characterization of the transition diagram is a straightforward task.

It is not difficult to see that the MDS-Reservation(t) queue has the following two key features: (a) any transition event changes the value of m by at most k , and (b) for $m \geq n - k + 1 + tk$, the transition from any state (w_1, m) to any other state $(w'_1, m' \geq n - k + 1 + tk)$ depends on $m \bmod k$ and not on the actual value of m . This results in a QBD process with boundary states and levels as specified in the statement of the theorem. Intuitively, this says that when $m \geq n - k + 1 + tk$, the presence of an additional batch at the end of the buffer has no effect on the functioning of the system. (In contrast, when $m < n - k + 1 + tk$, the system may behave differently if there was to be an additional batch, due to the possibility of this batch being within the threshold t . For instance, a job of this additional batch may be served upon completion of service at a server, which is not possible if this additional batch was not present).

■

Proof of Theorem 4: The MDS-Reservation(t) scheduling policy treats the first t waiting batches in the buffer as per the MDS scheduling policy, while imposing an additional restriction on batches $(t+1)$ onwards. When $t = \infty$, every batch is treated as in MDS, thus making MDS-Reservation(∞) identical to MDS. ■

Proof of Theorem 5: Under $M^k/M/n(0)$, any job can be processed by any server, and hence a server may be idle only when the buffer is empty. Thus, when $m \leq n$, all m jobs are in the servers, and the buffer is empty. When $m > n$, all the n servers are full and the remaining $(m - n)$ jobs are in the buffer. The transitions follow as a direct consequence of these observations. It also follows that when $m \leq n$, $b = 0$ and $z = n - m$. In addition, in state $m (> n)$ it must be that $w_1 = (m - n) \bmod k$, $b = \lceil \frac{m - n}{k} \rceil$, and for $i \in \{2, \dots, b\}$, $w_i = k$. Thus the knowledge of m suffices to describe the configuration of the entire system. ■

Proof of Theorem 6: For any state $(w_1, w_2, \dots, w_t, m)$, define q as in (8). The values of b , z , w_i , are identical to that in the proof of Theorem 3. Given the complete description of the state of the system as above, the

characterization of the transition diagram is a straightforward task. It is not difficult to see that the $M^k/M/n(t)$ queues have the following two key features: (a) any transition event changes the value of m by at most k , and (b) for $m \geq n+1+tk$, the transition from any state (w_1, m) to any other state $(w'_1, m' \geq n+1+tk)$ depends on $m \bmod k$ and not on the actual value of m . This results in a QBD process with boundary states and levels as specified in the statement of the theorem. Intuitively, this says that when $m \geq n+1+tk$, the total number of waiting batches is strictly greater than t . In this situation, the presence of an additional batch at the end of the buffer has no effect on the functioning of the system. ■

Proof of Theorem 7: The $M^k/M/n(t)$ scheduling policy follows the MDS scheduling policy when the number of batches in the buffer is less than or equal to t . Thus, $M^k/M/n(\infty)$ is always identical to MDS. ■

Proof of Theorem 8: In the MDS queue, suppose there are a large number of batches waiting in the buffer. Then, whenever a server completes a service, one can always find a waiting batch that has not been served by that server. Thus, no server is ever idle. Since the system has n servers, each serving jobs with times i.i.d. exponential with rates μ , the average number of jobs exiting the system per unit time is $n\mu$. The above argument also implies that under no circumstances (under any scheduling policy), can the average number of jobs exiting the system per unit time exceed $n\mu$. Finally, since each batch consists of k jobs, the rate at which batches exit the system is $\lambda_{MDS}^* = \frac{n\mu}{k}$ per unit time. Since the $M^k/M/n(t)$ queues upper bound the performance of the MDS queue, $\lambda_{M^k/M/n(t)}^* = \frac{n\mu}{k}$ for every t .

We shall now evaluate the maximum throughput of MDS-Reservation(1) by exploiting properties of QBD systems. In general, the maximum throughput λ^* of any QBD system is the value of λ such that: $\exists \mathbf{v}$ satisfying $\mathbf{v}^T(A_0 + A_1 + A_2) = 0$ and $\mathbf{v}^T A_0 \mathbf{1} = \mathbf{v}^T A_2 \mathbf{1}$, where $\mathbf{1} = [1 \ 1 \ \dots \ 1]^T$. Note that the matrices A_0 , A_1 and A_2 are affine transformations of λ (for fixed values of μ and k). Using the values of A_0 , A_1 , A_2 in the QBD representation of MDS-Reservation(1), we can show that $\lambda_{\text{Resv}(1)}^* \geq (1 - O(n^{-2})) \frac{n}{k} \mu$. For $t \geq 2$, each of the MDS-Reservation(t) queues upper bound MDS-Reservation(1), and are themselves upper bounded by the MDS queue. It follows that $\frac{n}{k} \mu \geq \lambda_{\text{Resv}(t)}^* \geq (1 - O(n^{-2})) \frac{n}{k} \mu$ for $t \geq 1$.

The value of $\lambda_{\text{Resv}(t)}^*$ can be explicitly computed for any value of n , k and t via the method described above. We perform this computation for $k=2$ and $k=3$ when $t=1$ to obtain the result mentioned in the statement of the theorem. We show the computation for $k=2$ here.

When $k=2$ and $t=1$, the j^{th} level of the QBD process consists of states $\{0, 1, 2\} \times \{n-1+2j, n+2j\}$ for $j \geq 1$. However, as seen in Fig. 7, several of these states never occur. In particular, in level j , only the states $(1, n-1+2j)$, $(1, n+2j)$ and $(2, n+2j)$ may be visited. Thus, to simplify notation, in the following discussion we consider the QBD process assuming the existence of only these three states (in that order) in every level. Under this representation, we have

$$A_0 = \begin{bmatrix} 0 & \mu & (n-1)\mu \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad A_1 = \begin{bmatrix} -n\mu - \lambda & 0 & 0 \\ (n-1)\mu & -(n-1)\mu - \lambda & 0 \\ n\mu & 0 & -n\mu - \lambda \end{bmatrix}, \quad A_2 = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix}.$$

$$\Rightarrow A_0 + A_1 + A_2 = \begin{bmatrix} -n & 1 & n-1 \\ n-1 & -(n-1) & 0 \\ n & 0 & -n \end{bmatrix} \mu \quad (13)$$

One can verify that the vector

$$\mathbf{v} = [v_1 \ v_2 \ v_3]^T = \left[n-1 \quad 1 \quad \frac{(n-1)^2}{n} \right]^T$$

satisfies

$$\mathbf{v}^T (A_0 + A_1 + A_2) = 0. \quad (14)$$

Thus,

$$\mathbf{v}^T A_0 \mathbf{1} = n(n-1)\mu, \quad (15)$$

and

$$\mathbf{v}^T A_2 \mathbf{1} = \lambda \left(n + \frac{(n-1)^2}{n} \right). \quad (16)$$

According to the properties of QBD processes, the value of $\lambda = \lambda_{\text{Resv}(1)}^*$ must satisfy $\mathbf{v}^T A_0 \mathbf{1} = \mathbf{v}^T A_2 \mathbf{1}$. Thus,

$$\lambda_{\text{Resv}(1)}^* = \frac{n^2(n-1)}{2n^2-2n+1} = \left(1 - \frac{1}{2n^2-2n+1} \right) \frac{n}{2} \mu. \quad (17)$$

■